Complete Guide to Java Threads: SAUCE

yatlas

1 Introduction to Threads

1.1 What is a Thread?

Imagine you're cooking dinner. You might be:

- Boiling pasta (main task)
- Occasionally stirring a sauce (secondary task)
- Checking the oven (another task)

You're doing multiple things *concurrently* - switching between tasks to make progress on all of them. A **thread** is like one of these cooking tasks.

In computing terms:

- A **thread** is a single sequence of executed instructions
- Your program can have multiple threads running **concurrently**
- Each thread has its own path of execution

1.2 Why Use Threads?

Threads help us:

- Keep applications responsive (UI doesn't freeze during long operations)
- Utilize multiple CPU cores
- Handle multiple operations simultaneously (download files while user interacts)
- Improve performance for certain types of tasks

2 Creating Your First Threads

2.1 Method 1: Extending Thread Class

```java
// Create a thread by extending Thread class
class MyThread extends Thread {
    private String threadName;

    public MyThread(String name) {
        this.threadName = name;
    }

    // This method runs when thread starts
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + " - Count: " + i);
            try {
                Thread.sleep(1000); // Pause for 1 second
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted");
            }
        }
        System.out.println(threadName + " finished");
    }
}

public class FirstThreadExample {
    public static void main(String[] args) {
        System.out.println("Main thread starting");

        // Create thread objects
        MyThread thread1 = new MyThread("Thread 1");
        MyThread thread2 = new MyThread("Thread 2");

        // Start the threads (don't call run() directly!)
        thread1.start();
        thread2.start();

        // Main thread continues executing
        System.out.println("Main thread continuing work");
    }
}
```

Listing 1: Simple Thread Example

When you run this, you'll see output like:

```
Main thread starting
Main thread continuing work
Thread 1 - Count: 1
Thread 2 - Count: 1
Thread 1 - Count: 2
Thread 2 - Count: 2
...
```

Notice how the threads run **concurrently**, not necessarily in order.

2.2 Method 2: Implementing Runnable Interface

```java
// Create a thread by implementing Runnable
class MyRunnable implements Runnable {
```

```
    private String threadName;

    public MyRunnable(String name) {
        this.threadName = name;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(threadName + " - Count: " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted");
            }
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        System.out.println("Main thread starting");

        // Create Runnable objects
        MyRunnable runnable1 = new MyRunnable("Runnable 1");
        MyRunnable runnable2 = new MyRunnable("Runnable 2");

        // Create Thread objects with our Runnables
        Thread thread1 = new Thread(runnable1);
        Thread thread2 = new Thread(runnable2);

        // Start the threads
        thread1.start();
        thread2.start();

        System.out.println("Main thread finished");
    }
}
```

Listing 2: Using Runnable Interface

2.3 Which Method to Use?

- **Prefer Runnable** - It's more flexible (you can extend other classes)
- Use **Thread extension** only when you need to override Thread methods
- Most professional code uses **Runnable** or **Lambda expressions**

2.4 Using Lambda Expressions (Java 8+)

```
public class LambdaThreads {
    public static void main(String[] args) {
        // Simple thread with lambda
        Thread thread1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Lambda thread - Count: " + i);
```

```
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {}
            }
        });

        thread1.start();

        // Even shorter with method reference
        Thread thread2 = new Thread(LambdaThreads::doWork);
        thread2.start();
    }

    private static void doWork() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Method reference - Count: " + i);
            try { Thread.sleep(1000); }
            catch (InterruptedException e) {}
        }
    }
}
```

Listing 3: Threads with Lambdas

3 Understanding Thread States

Threads go through different states during their lifetime:

```
public class ThreadStatesDemo {
    public static void main(String[] args) throws InterruptedException
    {
        Thread thread = new Thread(() -> {
            try {
                System.out.println("Thread started, state: " +
                    Thread.currentThread().getState());
                Thread.sleep(2000);
                synchronized(ThreadStatesDemo.class) {
                    ThreadStatesDemo.class.wait();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        System.out.println("Before start: " + thread.getState()); //
    NEW

        thread.start();
        Thread.sleep(100);
        System.out.println("After start: " + thread.getState());  //
    RUNNABLE or TIMED_WAITING

        Thread.sleep(100);
        System.out.println("During sleep: " + thread.getState()); //
    TIMED_WAITING

        Thread.sleep(2000);
        synchronized(ThreadStatesDemo.class) {
```

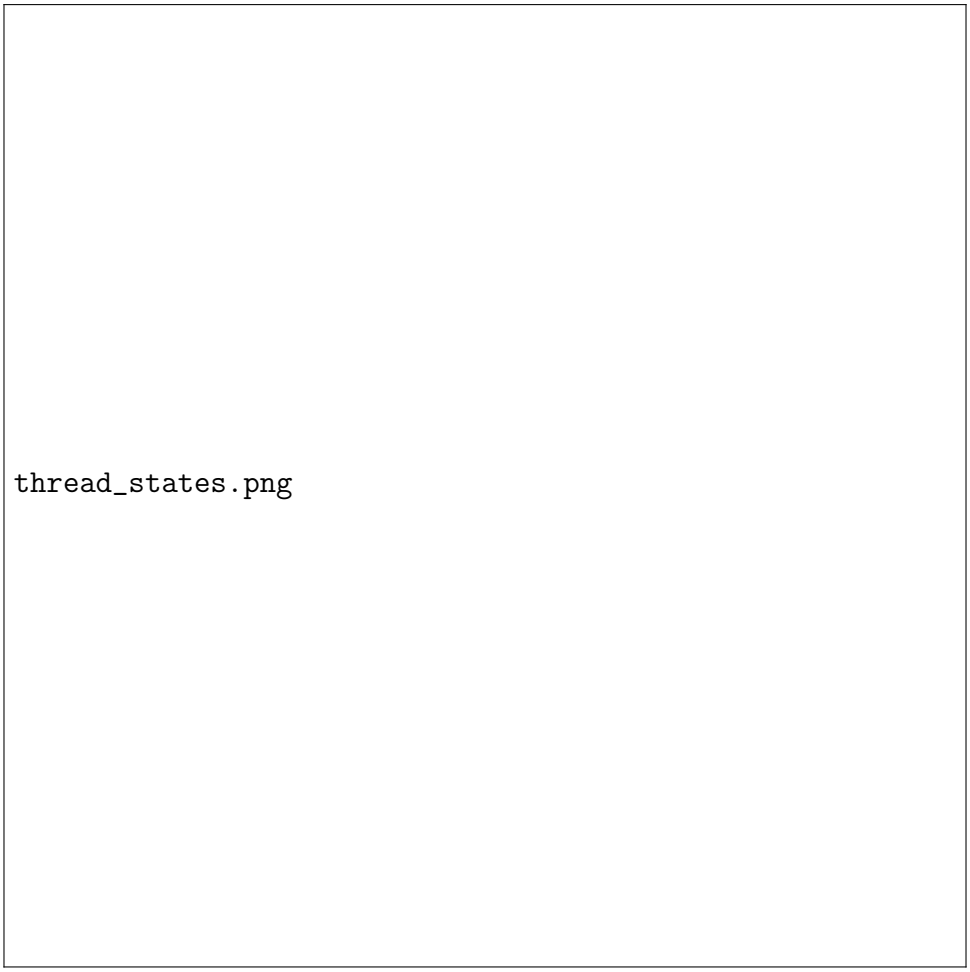

Figure 1: Thread Lifecycle States

```
27              System.out.println("After sleep: " + thread.getState()); //
    WAITING
28              ThreadStatesDemo.class.notify();
29          }
30
31          thread.join();
32          System.out.println("After completion: " + thread.getState());
    // TERMINATED
33      }
34  }
```

Listing 4: Observing Thread States

3.1 Thread State Definitions

- **NEW**: Created but not started
- **RUNNABLE**: Executing or ready to execute
- **BLOCKED**: Waiting for a monitor lock
- **WAITING**: Waiting indefinitely for another thread

- **TIMED_WAITING**: Waiting for a specified time
- **TERMINATED**: Finished execution

4 Thread Priorities and Daemon Threads

4.1 Thread Priorities

```
public class ThreadPriorityExample {
    public static void main(String[] args) {
        Thread lowPriority = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Low priority thread");
                Thread.yield(); // Hint to scheduler to switch threads
            }
        });

        Thread highPriority = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("High priority thread");
                Thread.yield();
            }
        });

        lowPriority.setPriority(Thread.MIN_PRIORITY);    // Priority 1
        highPriority.setPriority(Thread.MAX_PRIORITY);   // Priority 10

        lowPriority.start();
        highPriority.start();
    }
}
```

Listing 5: Thread Priorities

Important: Thread priorities are just hints to the OS scheduler. They're not guaranteed to work exactly as expected.

4.2 Daemon Threads

```
public class DaemonThreadExample {
    public static void main(String[] args) {
        // User thread (default)
        Thread userThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("User thread working...");
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {}
            }
            System.out.println("User thread finished");
        });

        // Daemon thread
        Thread daemonThread = new Thread(() -> {
            while (true) {
                System.out.println("Daemon thread running...");
```

```
                try { Thread.sleep(500); }
                catch (InterruptedException e) {}
            }
        });
        daemonThread.setDaemon(true); // Mark as daemon thread

        userThread.start();
        daemonThread.start();

        // When main and userThread finish, daemonThread will be
    terminated
        // even if it's in the middle of its work
    }
}
```

Listing 6: Daemon Threads

Key Points about Daemon Threads:

- JVM exits when only daemon threads remain
- Use for background tasks that can be safely abandoned
- Don't use for critical work (file saving, database commits)

5 Basic Thread Synchronization

5.1 The Problem: Why We Need Synchronization

```
class Counter {
    private int count = 0;

    public void increment() {
        count++; // This is the problem line
    }

    public int getCount() {
        return count;
    }
}

public class SynchronizationProblem {
    public static void main(String[] args) throws InterruptedException
    {
        Counter counter = new Counter();

        // Create multiple threads that increment the same counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
```

```
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        // We expect 2000, but often get less!
        System.out.println("Final count: " + counter.getCount());
    }
}
```

Listing 7: The Shared Resource Problem

The problem is that `count++` is actually three operations:

1. Read current value
2. Increment value
3. Write new value

Two threads can both read the same value, both increment, and both write - resulting in lost increments.

5.2 Solution: synchronized keyword

```
class SafeCounter {
    private int count = 0;

    // synchronized ensures only one thread can execute this method at a time
    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}

// Alternative: synchronized block
class AnotherSafeCounter {
    private int count = 0;
    private final Object lock = new Object(); // dedicated lock object

    public void increment() {
        synchronized(lock) { // Only one thread can enter this block per lock object
            count++;
        }
    }

    public int getCount() {
        synchronized(lock) {
            return count;
        }
```

```
    }
}
```

Listing 8: Using synchronized

5.3 Instance vs Class Level Synchronization

```
class MixedCounter {
    private int instanceCount = 0;
    private static int classCount = 0;

    // Synchronizes on the instance (this)
    public synchronized void incrementInstance() {
        instanceCount++;
    }

    // Synchronizes on the class (MixedCounter.class)
    public static synchronized void incrementClass() {
        classCount++;
    }

    // What this actually means:
    public void incrementInstanceEquivalent() {
        synchronized(this) { // Lock on current object
            instanceCount++;
        }
    }

    public static void incrementClassEquivalent() {
        synchronized(MixedCounter.class) { // Lock on class object
            classCount++;
        }
    }
}

public class SyncLevelExample {
    public static void main(String[] args) {
        MixedCounter counter1 = new MixedCounter();
        MixedCounter counter2 = new MixedCounter();

        // These don't block each other - different instances
        new Thread(() -> counter1.incrementInstance()).start();
        new Thread(() -> counter2.incrementInstance()).start();

        // These DO block each other - same class lock
        new Thread(() -> MixedCounter.incrementClass()).start();
        new Thread(() -> MixedCounter.incrementClass()).start();
    }
}
```

Listing 9: Instance vs Class Synchronization

6 wait() and notify() Methods

6.1 Basic Producer-Consumer Pattern

```java
class Message {
    private String message;
    private boolean empty = true;

    public synchronized String read() {
        // Wait until message is available
        while (empty) {
            try {
                wait(); // Releases lock and waits
            } catch (InterruptedException e) {}
        }
        empty = true;
        notifyAll(); // Notify waiting threads
        return message;
    }

    public synchronized void write(String message) {
        // Wait until message has been read
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        empty = false;
        this.message = message;
        notifyAll(); // Notify waiting threads
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
        Message message = new Message();

        // Producer thread
        Thread producer = new Thread(() -> {
            String[] messages = {"Message 1", "Message 2", "Message 3", "FINISH"};
            for (String msg : messages) {
                message.write(msg);
                System.out.println("Produced: " + msg);
                try { Thread.sleep(1000); }
                catch (InterruptedException e) {}
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() -> {
            for (String msg = message.read(); !msg.equals("FINISH"); msg = message.read()) {
                System.out.println("Consumed: " + msg);
                try { Thread.sleep(1500); }
                catch (InterruptedException e) {}
            }
        });

        producer.start();
        consumer.start();
    }
```

```
}
```

Listing 10: Simple Producer-Consumer

6.2 Understanding wait() and notify()

- **wait()**: Releases the lock and puts thread to sleep until notified
- **notify()**: Wakes up one waiting thread
- **notifyAll()**: Wakes up all waiting threads
- Must be called from **synchronized** context
- Always use **while loop** to check condition, not **if**

7 Common Thread Operations

7.1 Joining Threads

```
public class JoinExample {
    public static void main(String[] args) throws InterruptedException
   {
        Thread slowThread = new Thread(() -> {
            System.out.println("Slow thread starting");
            try { Thread.sleep(3000); }
            catch (InterruptedException e) {}
            System.out.println("Slow thread finished");
        });

        Thread fastThread = new Thread(() -> {
            System.out.println("Fast thread starting");
            try { Thread.sleep(1000); }
            catch (InterruptedException e) {}
            System.out.println("Fast thread finished");
        });

        slowThread.start();
        fastThread.start();

        System.out.println("Main thread waiting for fast thread...");
        fastThread.join(); // Main thread waits for fastThread to
   finish

        System.out.println("Main thread waiting for slow thread...");
        slowThread.join(2000); // Wait max 2 seconds

        System.out.println("Main thread continuing");
    }
}
```

Listing 11: Using join()

7.2 Thread Interruption

```
public class InterruptionExample {
    public static void main(String[] args) throws InterruptedException
    {
        Thread worker = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Working...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Thread interrupted during sleep
    ");
                    Thread.currentThread().interrupt(); // Restore
    interrupt flag
                }
            }
            System.out.println("Thread finished gracefully");
        });

        worker.start();
        Thread.sleep(3500); // Let worker run for a bit
        worker.interrupt(); // Politely ask thread to stop
        worker.join();      // Wait for thread to finish
    }
}
```

Listing 12: Thread Interruption

8 Practical Examples

8.1 File Download Manager

```
import java.util.*;
import java.util.concurrent.*;

public class DownloadManager {
    private final ExecutorService executor = Executors.
    newFixedThreadPool(3);
    private final List<Future<String>> downloads = new ArrayList<>();

    public void downloadFile(String url) {
        Callable<String> downloadTask = () -> {
            System.out.println("Starting download: " + url);
            // Simulate download time
            int downloadTime = ThreadLocalRandom.current().nextInt
    (1000, 5000);
            Thread.sleep(downloadTime);
            System.out.println("Finished download: " + url);
            return "Content of " + url;
        };

        Future<String> future = executor.submit(downloadTask);
        downloads.add(future);
    }

```

```
    public void shutdown() throws Exception {
        executor.shutdown();
        if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
            executor.shutdownNow();
        }

        // Collect results
        for (Future<String> download : downloads) {
            String result = download.get();
            System.out.println("Got result: " + result.substring(0,
    Math.min(20, result.length())));
        }
    }

    public static void main(String[] args) throws Exception {
        DownloadManager manager = new DownloadManager();

        manager.downloadFile("http://example.com/file1");
        manager.downloadFile("http://example.com/file2");
        manager.downloadFile("http://example.com/file3");
        manager.downloadFile("http://example.com/file4");

        Thread.sleep(2000); // Let downloads run

        manager.shutdown();
    }
}
```

Listing 13: Simple Download Manager

9 Best Practices and Common Pitfalls

9.1 Do's and Don'ts

Do:

- Use **Runnable** over extending **Thread**
- Use thread pools (**ExecutorService**) for multiple tasks
- Handle **InterruptedException** properly
- Use **synchronized** consistently for shared resources
- Prefer **notifyAll()** over **notify()** unless you're sure

Don't:

- Don't call **run()** directly - use **start()**
- Don't use **stop()**, **suspend()**, or **resume()** (deprecated)
- Don't synchronize on **String** literals or boxed primitives
- Don't ignore **InterruptedException**
- Don't create unlimited threads

9.2 Common Mistakes

```
public class CommonMistakes {
    // Mistake 1: Calling run() instead of start()
    public static void mistake1() {
        Thread t = new Thread(() -> System.out.println("Running"));
        t.run();  // WRONG - runs in current thread
        t.start(); // CORRECT - runs in new thread
    }

    // Mistake 2: Synchronizing on different objects
    public static void mistake2() {
        final Object lock1 = new Object();
        final Object lock2 = new Object();

        Thread t1 = new Thread(() -> {
            synchronized(lock1) {
                // Critical section
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized(lock2) {  // WRONG - different lock!
                // This doesn't provide mutual exclusion with t1
            }
        });
    }

    // Mistake 3: Using if instead of while with wait()
    public static void mistake3() {
        Object lock = new Object();
        boolean condition = false;

        // WRONG
        synchronized(lock) {
            if (!condition) {
                try { lock.wait(); } catch (InterruptedException e) {}
            }
            // Condition might be false again after wait!
        }

        // CORRECT
        synchronized(lock) {
            while (!condition) {  // Always recheck after wait
                try { lock.wait(); } catch (InterruptedException e) {}
            }
        }
    }
}
```

Listing 14: Common Thread Mistakes

10 Exercises for Practice

10.1 Exercise 1: Basic Thread Creation

Create three threads that print numbers 1-5 with their thread name. Make the main thread wait for all three to complete.

10.2 Exercise 2: Thread-Safe Bank Account

Create a `BankAccount` class with `deposit` and `withdraw` methods. Make it thread-safe using synchronization.

10.3 Exercise 3: Simple Task Scheduler

Create a task scheduler that can run tasks concurrently but limits to 3 concurrent tasks using a fixed thread pool.

10.4 Exercise 4: Producer-Consumer with Buffer

Implement a producer-consumer system where producers add items to a fixed-size buffer and consumers remove them. Handle the cases when the buffer is full or empty.

11 Next Steps

After mastering these basics, you can explore:

- **java.util.concurrent** package (advanced utilities)
- **Lock** objects and **Condition** variables
- **Atomic** variables for lock-free programming
- **Concurrent collections** (ConcurrentHashMap, CopyOnWriteArrayList)
- Thread pools and executors in more depth
- Performance optimization and debugging

Remember: Thread programming requires practice. Start simple, test thoroughly, and gradually build more complex systems.